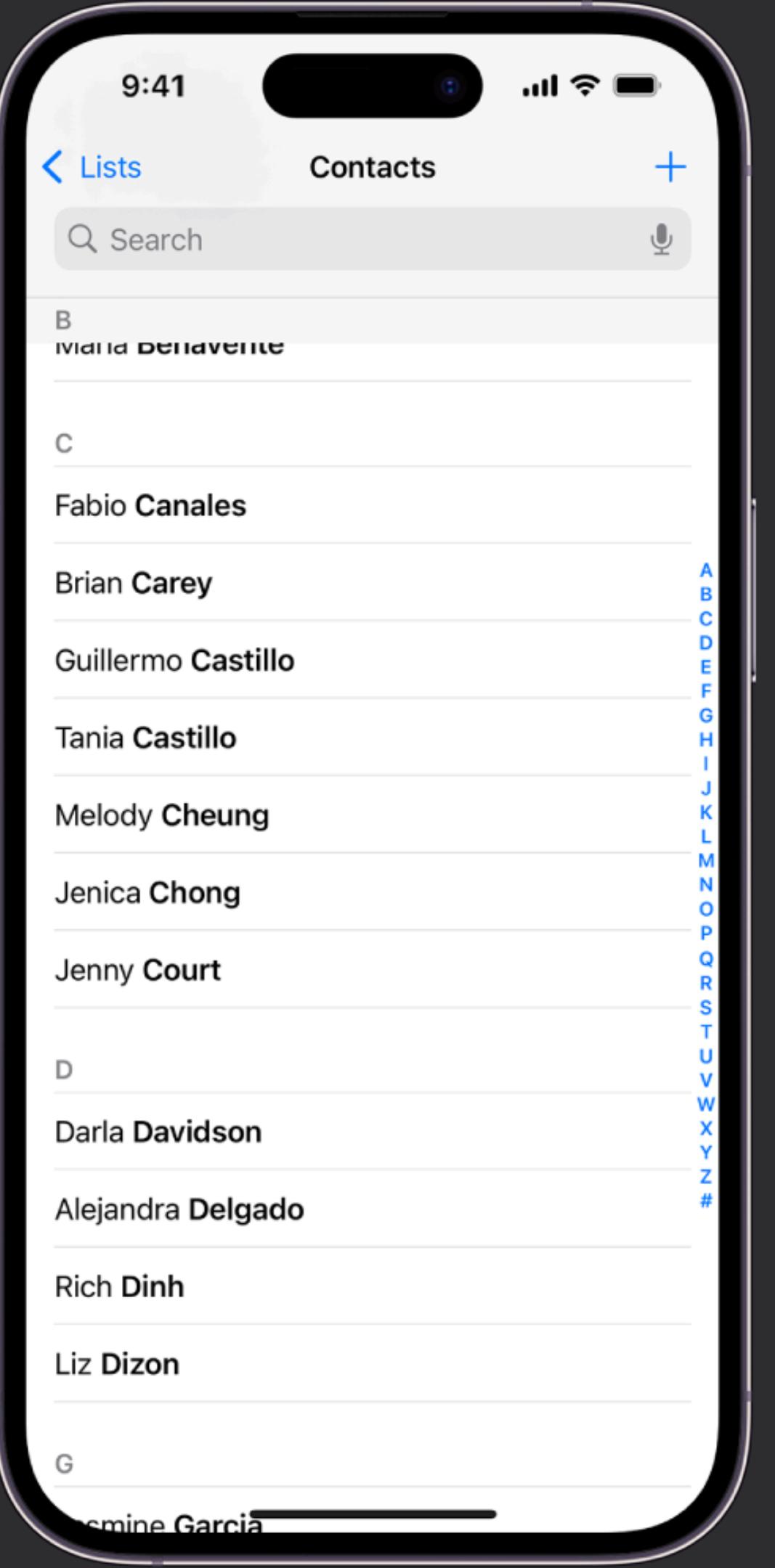
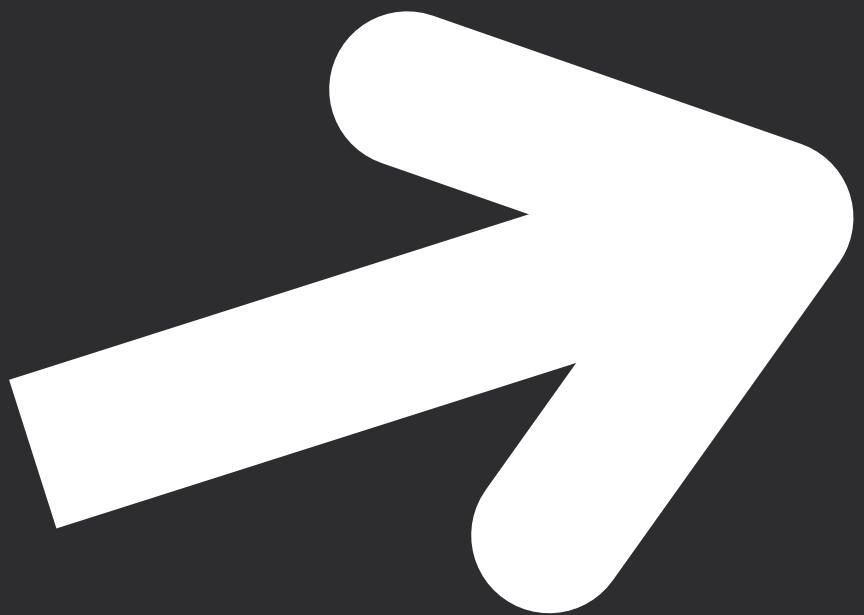


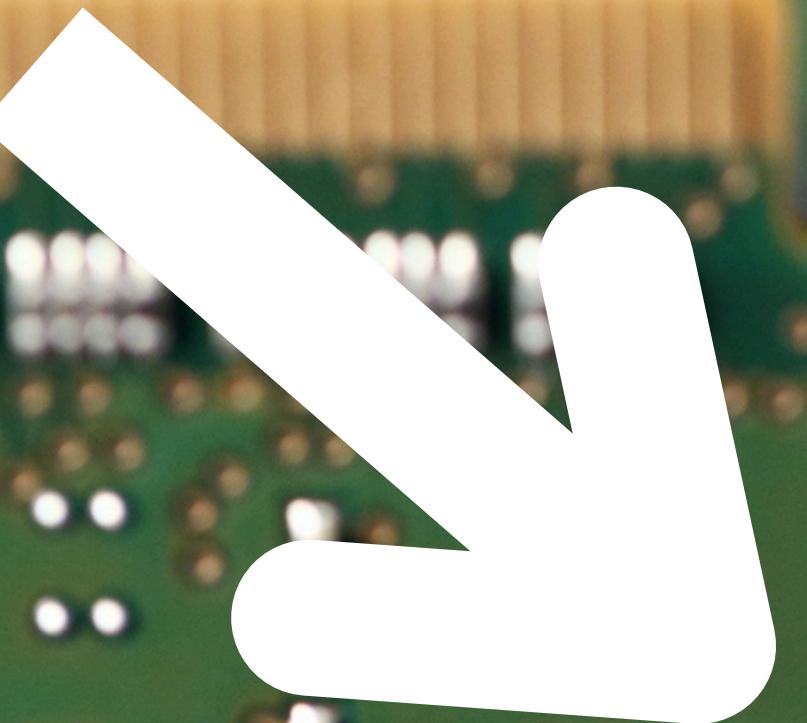
Complexidade em Algoritmos

Preciso encontrar uma **PALAVRA** em um
DICIONÁRIO

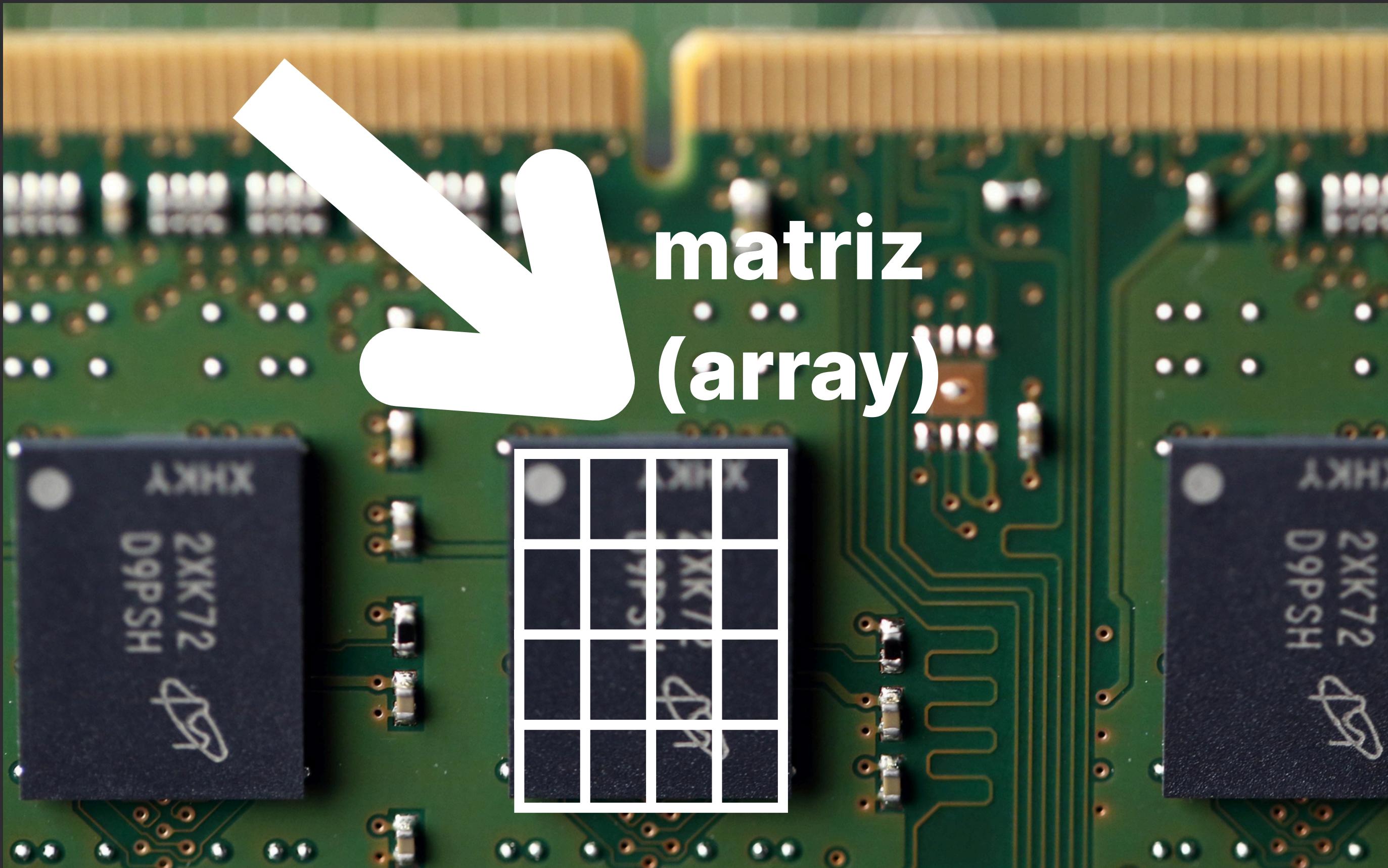




_aula 2 (aedii)



matriz
(array)



0	1	2	3
			n

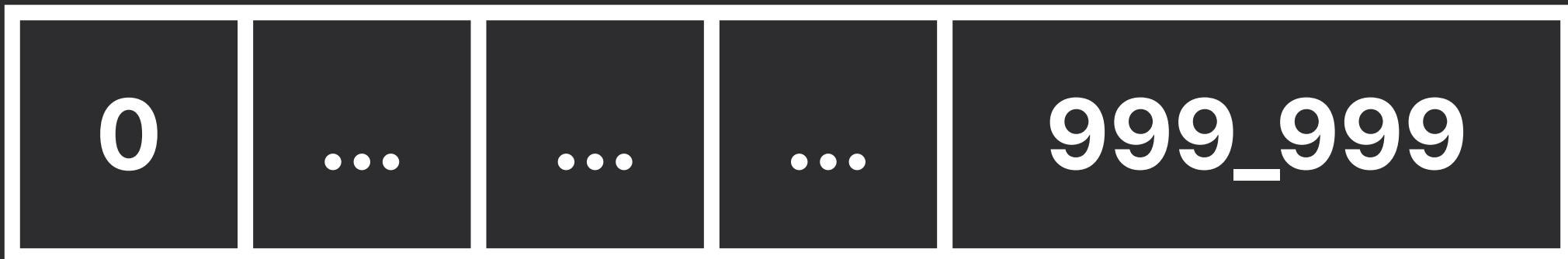
abstração de memória
matriz (array)

simplificando matriz (array)

0	1	2	3	4	5	6
---	---	---	---	---	---	---

$$n = 7$$

$n-1$ indica o último
elemento do array



$$n = (1 \text{ milhão} - 1)$$

lista de contatos

matriz (array)



$$n = 7$$

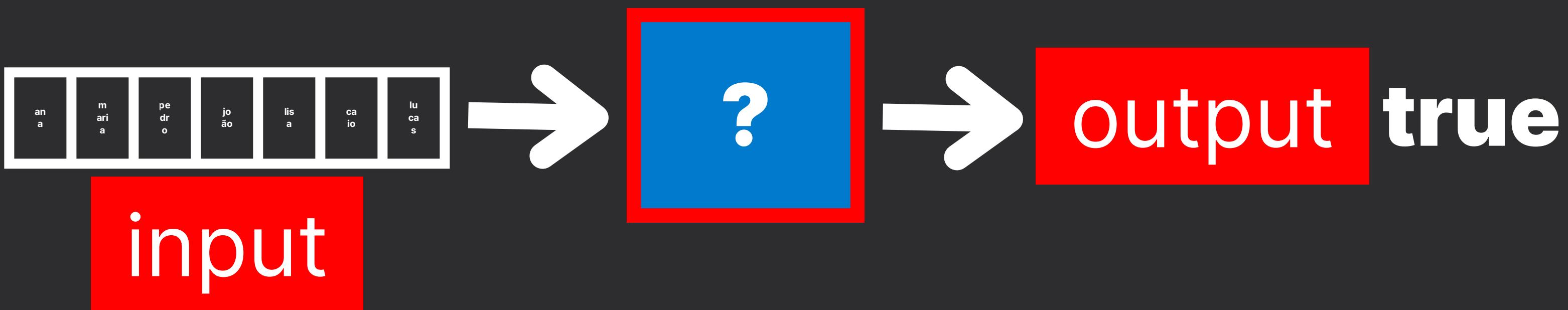
Preciso encontrar um **PALAVRA** nome em
uma ~~DICIONÁRIO~~ lista de contatos

existe `lucas` nos
contatos?

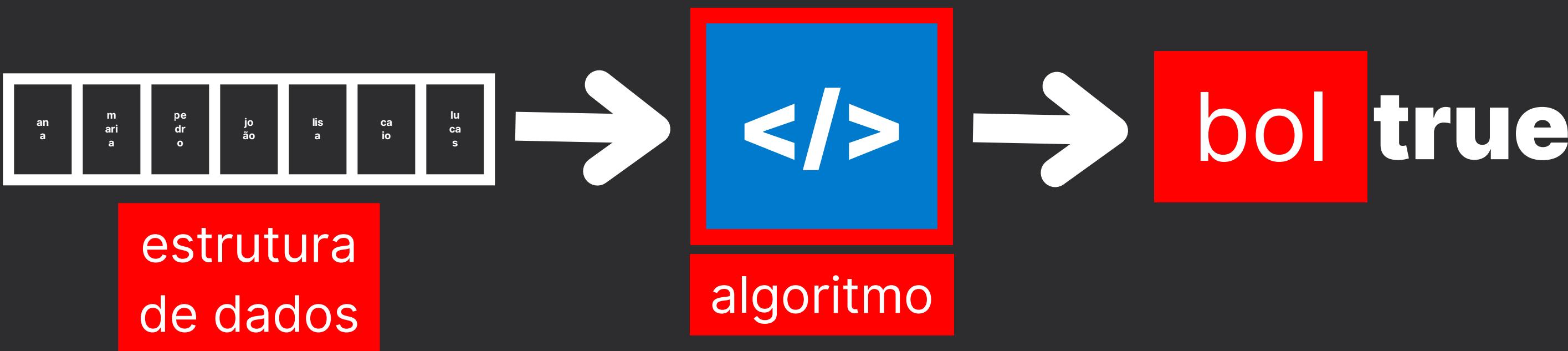


contatos[6] **true**

existe `lucas` nos
contatos?



existe `lucas` nos
contatos?



ana [0]	maria [1]	pedro [2]	joão [3]	lisa [4]	caio [5]	lucas [6]
------------	--------------	--------------	-------------	-------------	-------------	--------------



```
contatos = ["ana", "maria", "pedro", "joão",
"lisa", "caio", "lucas"]
def agenda():
    for i in contatos:
        if i == "lucas":
            print("Encontrado")
            return 0
    print("Não Encontrado")
    return 1
agenda()
```

ana [0]	maria [1]	pedro [2]	joão [3]	lisa [4]	caio [5]	lucas [6]
------------	--------------	--------------	-------------	-------------	-------------	--------------



```
contatos = ["ana", "maria", "pedro", "joão",
"lisa", "caio", "lucas"]
def agenda():
    for i in contatos:
        if i == "lucas":
            print("Encontrado")
            return 0
    print("Não Encontrado")
    return 1
agenda()
```

ana [0]	maria [1]	pedro [2]	joão [3]	lisa [4]	caio [5]	lucas [6]
------------	--------------	--------------	-------------	-------------	-------------	--------------



```
contatos = ["ana", "maria", "pedro", "joão",
"lisa", "caio", "lucas"]
def agenda():
    for i in contatos:
        if i == "lucas":
            print("Encontrado")
            return 0
    print("Não Encontrado")
    return 1
agenda()
```

ana [0]	maria [1]	pedro [2]	joão [3]	lisa [4]	caio [5]	lucas [6]
------------	--------------	--------------	-------------	-------------	-------------	--------------



```
contatos = ["ana", "maria", "pedro", "joão",
"lisa", "caio", "lucas"]
def agenda():
    for i in contatos:
        if i == "lucas":
            print("Encontrado")
            return 0
    print("Não Encontrado")
    return 1
agenda()
```

n tempo para solucionar ou n operações*



```
contatos = ["ana", "maria", "pedro", "joão",
"lisa", "caio", "lucas"]
def agenda():
    for i in contatos:
        if i == "lucas":
            print("Encontrado")
            return 0
    print("Não Encontrado")
    return 1
agenda()
```

crescimento n

Por definição, o tempo de execução aumenta na **mesma proporção** que os dados. Se os dados dobram de tamanho, o tempo também dobra.

qual paradigma aqui?

problema à solucionar



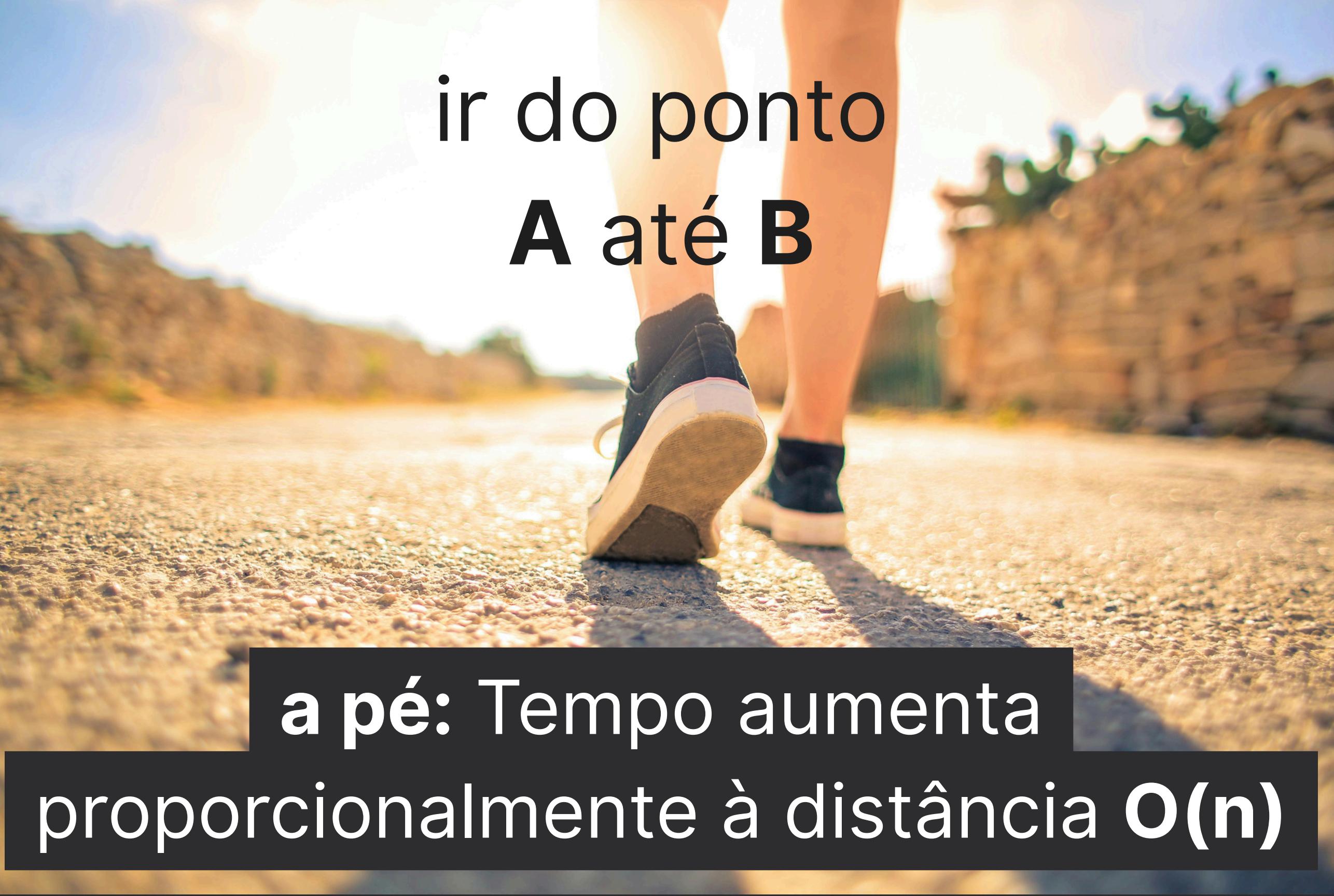
qual paradigma aqui?

problema à solucionar



BUSCA !

Na área da computação, uma forma de medir como algoritmos se comportam quando o volume de dados de entrada aumentam é a **Notação Big-O**



ir do ponto
A até B

a pé: Tempo aumenta
proporcionalmente à distância $O(n)$

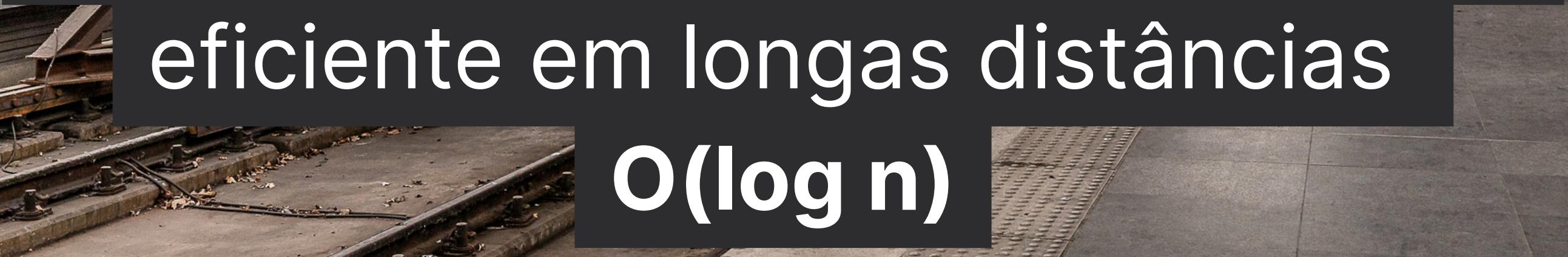


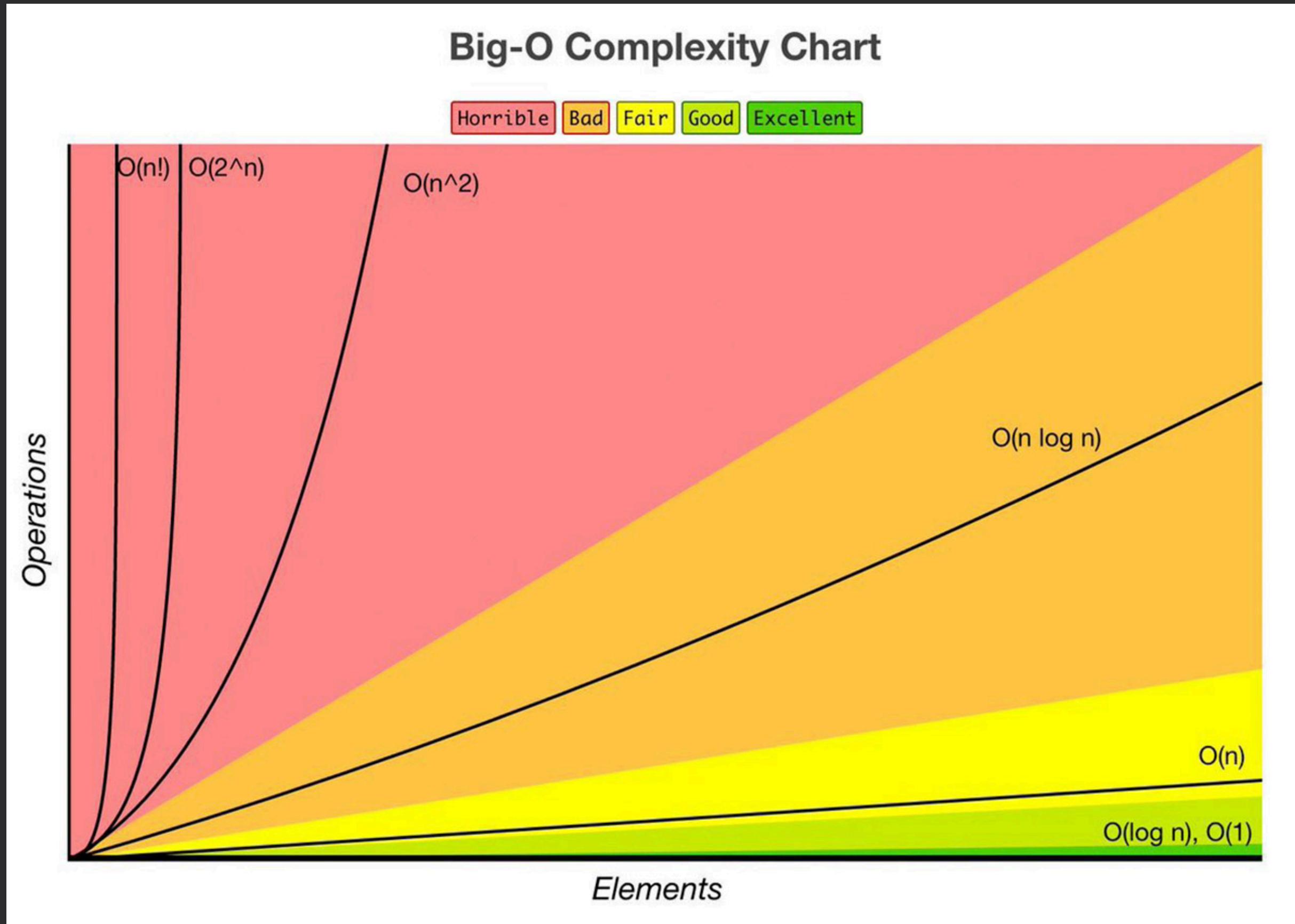
ir do ponto
A até B

de carro: Rápido em distâncias
curtas, mas pode enfrentar
congestionamentos $O(n^2)$



de trem: Tempo fixo de espera, mas
eficiente em longas distâncias
 $O(\log n)$





$O(n!)$

$O(n^2)$

$O(n \log n)$

$O(n)$ *linear search*

$O(\log n)$ *binary search*

$O(1)$



maior
complexidade

$O(1)$ - Complexidade Constante

Não importa quanto os dados aumentem,
o tempo de execução permanece
constante. É a complexidade ideal para
qualquer operação.

$O(\log n)$ - Complexidade Logarítmica

Quando os dados dobram de tamanho, o algoritmo só precisa de um pouco mais de tempo para executar. É eficiente para grandes volumes de dados.

$O(n)$ - Complexidade Linear

O tempo de execução aumenta na mesma proporção que os dados. Se os dados dobram de tamanho, o tempo também dobra.

$O(n \log n)$ - Complexidade Linearítmica

Um pouco mais lento que linear, mas muito melhor que quadrático. Representa o equilíbrio entre simplicidade e eficiência para muitos problemas.

$O(n^2)$ - Complexidade Quadrática

Muito mais lento quando os dados aumentam. Se os dados dobram de tamanho, o tempo de execução quadruplica.

É comum em algoritmos com loops aninhados.

Complexidade	Nome	10 itens	100 itens	1.000 itens	1.000.000 itens
 O(1)	Constante	1	1	1	1
 O(log n)	Logarítmica	3	7	10	20
 O(n)	Linear	10	100	1.000	1.000.000
 O(n log n)	Linearítmica	33	664	9.966	19.931.569
 O(n²)	Quadrática	100	10.000	1.000.000	1.000.000.000.000
 O(2^n)	Exponencial	1.024	1.267×10^{30}	1.07×10^{301}	Incomputável
● Excelente ● Use com cuidado ● Evite					

Valores aproximados em unidades de tempo relativas.

Um algoritmo $O(n^2)$ pode levar anos para processar 1 milhão de itens, enquanto um $O(n \log n)$ leva minutos.



```
# Loop aninhado: for dentro de outro for
for i in range(3): # loop externo
    for j in range(2): # loop interno
        print(f"i = {i}, j = {j}")
```

existe 50 nos valores?

100 [0]	5 [1]	20 [2]	10 [3]	1 [4]	500 [5]	50 [6]
------------	----------	-----------	-----------	----------	------------	-----------

```
i, arr = 0, [100,5,20,10,1,500,50]
while i < len(arr):
    if arr[i] == 50:
        print(f"Achei no índice {i}")
        break
    i+=1
if i == len(arr):
    print("Não achei")
```

O(n)
Complexidade
Linear

existe 50 nos valores?

1 [0]	5 [1]	10 [2]	20 [3]	50 [4]	100 [5]	500 [6]
----------	----------	-----------	-----------	-----------	------------	------------

```
i, arr = 0, [1,5,10,20,50,100,500]
while i < len(arr):
    if arr[i] == 50:
        print(f"Achei no índice {i}")
        break
    i+=1
if i == len(arr):
    print("Não achei")
```

O(n)
Complexidade
Linear ?

AINDA $O(n)$, porque por definição a notação Big-O mede o pior caso, não devemos contar com que em cenários reais, os dados irão favorecer o algoritmo implementado.

existe 50 nos valores?

1 [0]	5 [1]	10 [2]	20 [3]	50 [4]	100 [5]	500 [6]
----------	----------	-----------	-----------	-----------	------------	------------

como transformar o problema de $O(n)$
para um simples $O(\log n)$?

Complexidade Linear > Logarítmica

existe 50 nos valores?

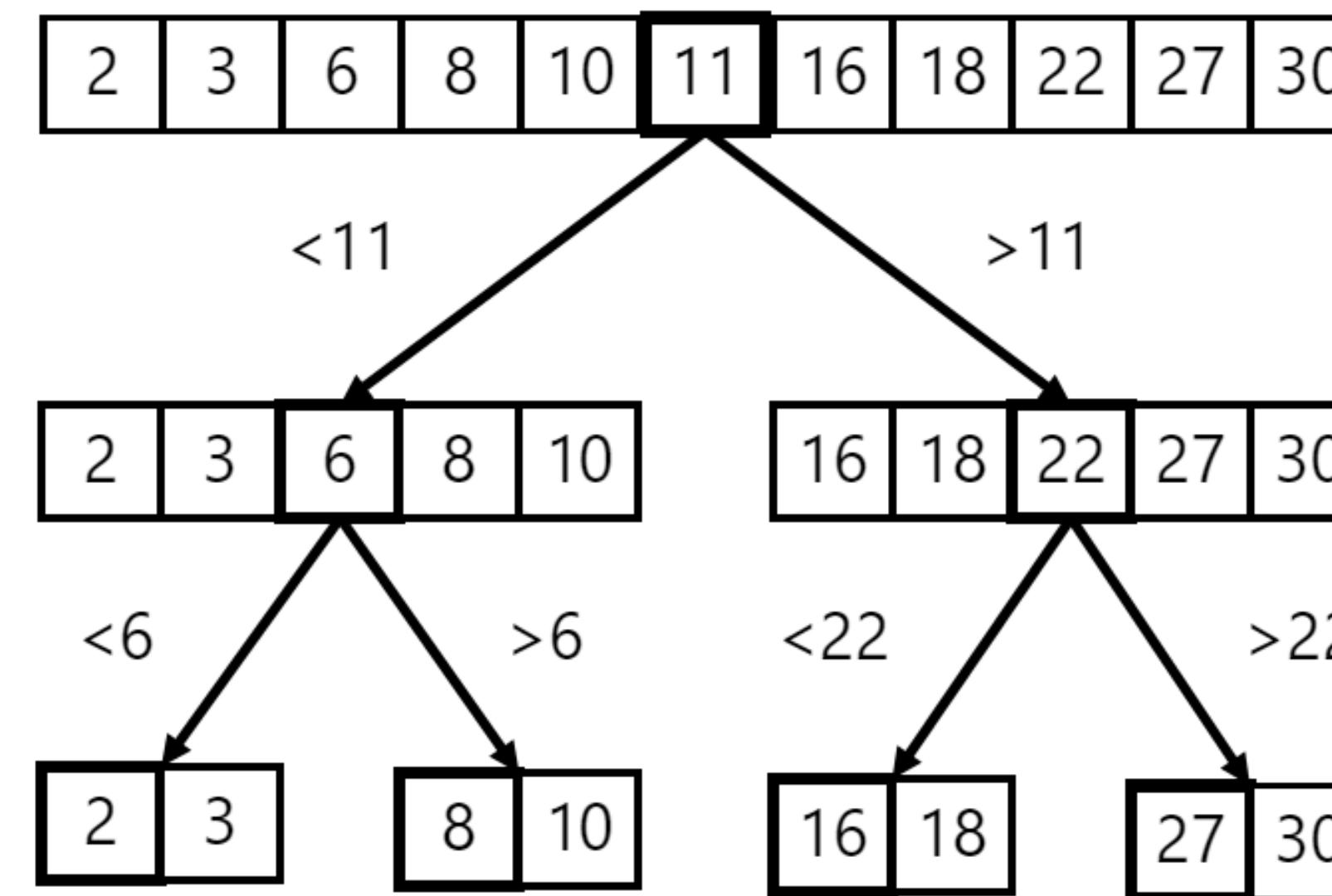
1 [0]	5 [1]	10 [2]	20 [3]	50 [4]	100 [5]	500 [6]
----------	----------	-----------	-----------	-----------	------------	------------

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2

        if arr[mid] == target:
            return mid # encontrado
        elif arr[mid] < target:
            left = mid + 1 # metade direita
        else:
            right = mid - 1 # metade esquerda
    return -1 # não encontrado
print(binary_search([1,5,10,20,50,100,500], 50))
```

O(log n)
busca binária

Binary Search



- Busca binária em arrays ordenados
- Busca em bancos de dados indexados
 - Algoritmos de divisão e conquista
 - Operações em árvores平衡adas



Dica de Ouro

Sempre que possível, ordene os dados antes de buscar.

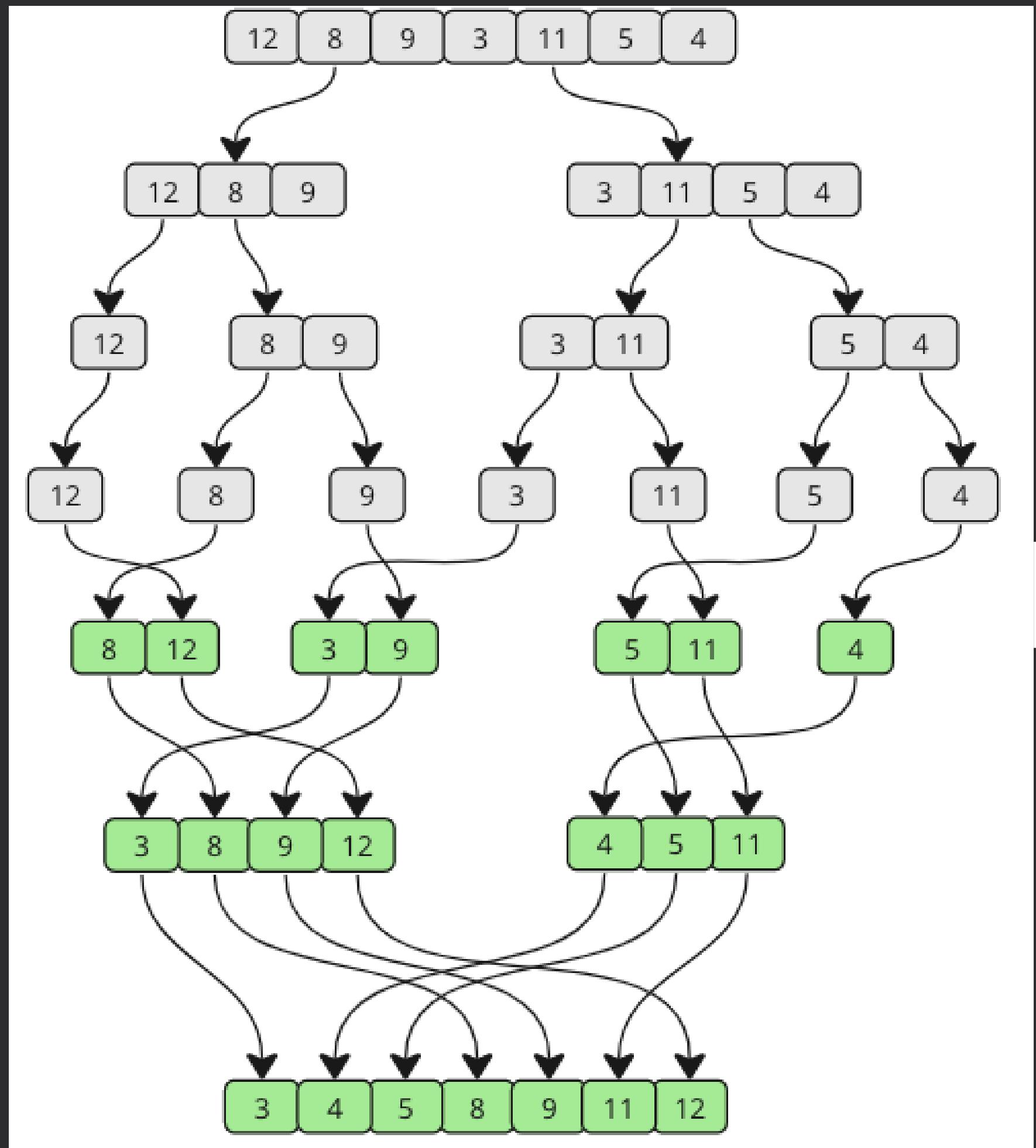
O custo inicial da ordenação $O(n \log n)$ é compensado por buscas extremamente rápidas do $O(\log n)$ posteriormente.

$O(\log n)$ ou $O(n)$?



```
# Encontrar o maior valor em um array
def find_max(arr):
    max_val = arr[0] # Assume o primeiro como maior
    # Percorre cada elemento do array
    for i in range(1, len(arr)):
        if arr[i] > max_val:
            max_val = arr[i] # Atualiza se encontrar maior
    return max_val
# Exemplo de uso
numbers = [4, 2, 9, 7, 5, 1]
print(find_max(numbers)) # Saída: 9
```





baralho desordenado

12 [0]	8 [1]	9 [2]	3 [3]	11 [4]	5 [5]	4 [6]
-----------	----------	----------	----------	-----------	----------	----------

aula 2 (aedii)

```

def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2 # Encontra o meio
        left_half = arr[:mid] # Divide em duas metades
        right_half = arr[mid:]

        # Recursivamente ordena as duas metades
        merge_sort(left_half)
        merge_sort(right_half)

    i = j = k = 0

    # Mescla as metades ordenadas
    while i < len(left_half) and j < len(right_half):
        if left_half[i] < right_half[j]:
            arr[k] = left_half[i]
            i += 1
        else:
            arr[k] = right_half[j]
            j += 1
        k += 1

    # Verifica elementos restantes
    while i < len(left_half):
        arr[k] = left_half[i]
        i += 1
        k += 1

    while j < len(right_half):
        arr[k] = right_half[j]
        j += 1
        k += 1

```



O(n log n)

O algoritmo *merge sort* é mais lento que linear, mas muito melhor que quadrático.

Envolve Recursividade

```

def bubble_sort(arr):
    n = len(arr)
    # Loop externo - percorre todo o array
    for i in range(n - 1):
        swapped = False
        # Loop interno - compara elementos adjacentes
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                # Troca os elementos se estiverem na ordem errada
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        # Se não houve trocas, o array já está ordenado
        if not swapped:
            break
    return arr # O(n²) - complexidade quadrática
# Exemplo de uso
baralho = [12,8,9,3,11,5,4]
print(bubble_sort(baralho))

```



O(n²) - Complexidade Quadrática

É comum em algoritmos com loops aninhados.

Complexidade em Algoritmos

parte 1

Complexidade em Algoritmos

parte 1